Społeczna Akademia Nauk w Łodzi

Programowanie współbieżne

Laboratorium 2

Prowadzący
mgr inż. Krystian Gumiński

2025

Lock

ReentrantLock to zaawansowany mechanizm synchronizacji w Javie, który działa podobnie do synchronized, ale daje większą kontrolę nad blokadą - można np. sprawdzić, czy lock jest dostępny (tryLock()), lub wymusić odblokowanie w finally.

Lock to interfejs definiujący podstawowe operacje blokowania i odblokowywania dostępu do zasobów współdzielonych między wątkami. tryLock() pozwala na "nieblokujące"podejście - próbuje zdobyć blokadę, ale jeśli nie jest dostępna, wątek może wykonać coś innego zamiast czekać. CountDownLatch to licznik, który pozwala wątkom czekać, aż inne zakończą określoną liczbę operacji (np. start równoczesny wątków po odliczeniu).

W kodzie (listing 1) utworzono dwa wątki (t1 i t2) oraz obiekt CountDownLatch gate, który służy do zsynchronizowanego startu. Każdy wątek próbuje zdobyć dwa locki (A i B), ale w odwrotnej kolejności:

- t1: najpierw blokuje A, potem próbuje zablokować B,
- t2: najpierw blokuje B, potem próbuje zablokować A.

Po wywołaniu gate.countDown() oba wątki ruszają jednocześnie. Ponieważ każdy z nich trzyma inny lock i próbuje zdobyć drugi, może dojść do zakleszczenia (deadlocka) - sytuacji, w której żaden z wątków nie może kontynuować, bo czeka na zasób zablokowany przez drugi. Wynik programu zaprezentowano na listingu 2

Listing 1: Przykład deadlocka

```
1       Thread t1 = new Thread(() -> {
2               try {
3                   gate.await();
4                   A.lock();
5                   System.out.println("MAM LOCKA A " + Thread.
                        currentThread().getName());
6                   sleep(100);
7                   System.out.println("Probuje wziac locka B...." +
                        Thread.currentThread().getName());
8                   B.lock();
9               } catch (InterruptedException e) {
10                  throw new RuntimeException(e);
11              } finally {
12                  A.unlock();
13                  B.unlock();
14              }
15          }, "T1");
16
17          Thread t2 = new Thread(() -> {
18              try {
19                  gate.await();
20                  B.lock();
```

```
21                  System.out.println("MAM LOCKA B " + Thread.
                        currentThread().getName());
22                  sleep(100);
23                  System.out.println("Probuje wziac locka A..... " +
                        Thread.currentThread().getName());
24                  A.lock();
25              } catch (InterruptedException e) {
26                  throw new RuntimeException(e);
27              } finally {
28                  B.unlock();
29                  A.unlock();
30              }
31          }, "T2");
32
33          t1.start();
34          t2.start();
35          gate.countDown();
36          t1.join();
37          t2.join();
```

Listing 2: Wynik programu - deadlock

```
1 MAM LOCKA A T1
2 MAM LOCKA B T2
3 Probuje wziac locka A..... T2
4 Probuje wziac locka B.... T1
```

Filozofowie

Problem ucztujących filozofów (ang. Dining Philosophers Problem) to klasyczny przykład z dziedziny programowania współbieżnego, pokazujący trudności w synchronizacji wielu wątków rywalizujących o ograniczone zasoby.

Wyobraźmy sobie grupę filozofów siedzących przy okrągłym stole. Między każdą parą filozofów leży jeden widelec (czyli zasób współdzielony). Aby jeść, filozof potrzebuje dwóch widelców: lewego i prawego. Jeśli jednak każdy filozof jednocześnie podniesie swój lewy widelec i będzie czekał na prawy, żaden nie będzie mógł kontynuować: nastąpi zakleszczenie (deadlock).

Przedstawiony program (listing 3) symuluje klasyczny problem ucztujących filozofów z wykorzystaniem mechanizmów synchronizacji w Javie. Każdy filozof jest reprezentowany przez osobny wątek, który próbuje zdobyć dwa zasoby: dwa widelce, odpowiadające obiektom typu ReentrantLock. Każdy z filozofów posiada dostęp do dwóch sąsiednich widelców: lewego i prawego, które są współdzielone z jego sąsiadami przy stole.

W metodzie run() każdy filozof najpierw czeka na sygnał startowy z obiektu CountDownLatch, aby wszyscy rozpoczęli działanie jednocześnie. Następnie próbuje zablokowa-

ć/podnieść swój lewy widelec, po czym robi krótką przerwę (Thread.sleep(100)), aby dać sąsiadom szansę na zrobienie tego samego. Po chwili każdy z nich próbuje zdobyć również prawy widelec, aby móc rozpocząć jedzenie. Gdyby udało się uzyskać oba zasoby, filozof wypisałby komunikat o rozpoczęciu jedzenia, a następnie zwolniłby oba widelce w blokach finally.

W praktyce jednak program nigdy nie dociera do momentu uczty. Każdy filozof z powodzeniem blokuje swój lewy widelec, ale gdy próbuje zablokować prawy, okazuje się, że ten jest już zajęty przez sąsiada. W efekcie żaden z wątków nie jest w stanie przejść dalej, mimo że wszystkie działają równocześnie i poprawnie używają mechanizmów blokujących. Wynik programu został zaprezentowany na listningu 4.

Listing 3: Filozofowie z deadllockiem

```
1  static class Philosopher implements Runnable {
2          private final int id;
3          private final ReentrantLock left, right;
4          private final CountDownLatch startGate;
5
6          Philosopher(int id, ReentrantLock left, ReentrantLock
               right, CountDownLatch startGate) {
7              this.id = id;
8              this.left = left;
9              this.right = right;
10             this.startGate = startGate;
11         }
12
13         @Override
14         public void run() {
15             try {
16                 startGate.await();
17                 // CEL: deadlock - wszyscy lapia NAJPIERW LEWY,
                       POTEM PRAWY
18                 System.out.println("Filozof " + id + " bierze lewy
                       widelec");
19                 left.lock();
20                 try {
21                     Thread.sleep(100);  // daj sasiadowi zlapac
                           jego lewy
22                     right.lock();       // tu wszyscy czekamy
23                     try {
24                         System.out.println("Filozof" + id + " je (
                               nie zobaczysz tego bo nigdy tu nie
                               dojdzie)");
25                     } finally {
26                         right.unlock();
```

```
27                      }
28                  } finally {
29                      left.unlock();
30                  }
31              } catch (InterruptedException ignored) {
32              }
33          }
34      }
35
36      public static void main(String[] args) throws
            InterruptedException {
37          int N = 5;
38          ReentrantLock[] forks = new ReentrantLock[N];
39          for (int i = 0; i < N; i++) forks[i] = new ReentrantLock()
                ; // fair=false wystarczy
40
41          CountDownLatch startGate = new CountDownLatch(1);
42          Thread[] phils = new Thread[N];
43
44          for (int i = 0; i < N; i++) {
45              ReentrantLock left = forks[i];
46              ReentrantLock right = forks[(i + 1) % N];
47              phils[i] = new Thread(new Philosopher(i, left, right,
                    startGate), "P" + i);
48              phils[i].start();
49          }
50
51          startGate.countDown();
52          for (Thread t : phils) t.join(); // program wisi -
                deadlock
53      }
```

Listing 4: Wynik programu - filozofowie

```
1 Filozof 0 bierze lewy widelec
2 Filozof 3 bierze lewy widelec
3 Filozof 2 bierze lewy widelec
4 Filozof 4 bierze lewy widelec
5 Filozof 1 bierze lewy widelec
```

1 Zadanie

Zaimplementować poprawne (bez zakleszczeń i bez zagłodzenia) rozwiązanie problemu ucztujących filozofów, uruchomić 30-sekundową symulację, zebrać statystyki oraz przygotować sprawozdanie ze screenami potwierdzającymi działanie.

1. Liczba filozofów: domyślnie 5
2. Czas trwania eksperymentu: dokładnie 30 s. Program ma zakończyć się samoczynnie po upływie czasu i wypisać podsumowanie.
3. Brak zakleszczeń (deadlock-free) i brak zagłodzenia (starvation-free) - uzasadnij w sprawozdaniu jaki mechanizm gwarantuje te własności.

Wymóg 1.1. Po zakończeniu symulacji program ma wydrukować zestawienie dla każdego filozofa:

- liczba cykli "myślenia",
- liczba cykli "jedzenia",

Przykładowe statystyki np.: F0 Myślał 120, Jadł: 118, F1 Myślał: 30, Jadł:42, ...